CS477 - Formal Software Development Methods

Verification of Sequential Programs

Grigore Roşu

(slides from José Meseguer)

Department of Computer Science
University of Illinois at Urbana-Champaign

Verification of Imperative Sequential Programs

We are now ready to consider the verification of sequential imperative programs. We will do so using a subset of the Java programming language.

Of course, for the formal verification of some properties Q about a program P in a sequential imperative language $\mathcal{L}$ to be meaningful at all, our first and most crucial task is to make sure that the programming language $\mathcal{L}$ has a clear and precise mathematical semantics, since only then can we settle mathematically whether a program P satisfies some properties Q.

Verification of Imperative Sequential Programs (II)

The issue of giving a mathematical semantics to a programming language $\mathcal{L}$ is actually nontrivial, particularly for imperative languages; it is of course much easier for a declarative language, since we can rely on the underlying logic on which such a language is based.

For example, for a Maude functional module, its mathematical semantics is given by the initial algebra of its equational theory, whereas its operational semantics is based on equational simplification with its equations, which are assumed confluent and terminating.

Some imperative languages have never been given a precise semantics; their only precise documentation may be the different compilers, perhaps inconsistent with each other.

Verification of Imperative Sequential Programs (III)

Of course, there is no reason why we cannot, if we try, give a precise mathematical semantics to a programming language. If the language is very baroque, this may of course be a big task; and if some dark corners are undocumented, a rational reconstruction of the language, making precise what nobody made clear before and perhaps got the compiler writers into trouble, may be needed. But it can be done.

Those who think otherwise, or who would prefer to live with ambiguous programming languages engage in a form of anti-mathematical irrationalism. A quite different question is whether a specific programming language is worth the effort of giving it a mathematical semantics.

Verification of Imperative Sequential Programs (IV)

In the end, giving mathematical semantics to a programming language $\mathcal{L}$ amounts to giving a mathematical model of the language. This is typically done using some mathematical formalism: either the language of set theory, which is a de-facto universal formalism for mathematics, or some other well-defined formalism.

For sequential imperative languages equational formalisms are quite well-suited to the task. In traditional denotational semantics, a higher-order equational logic, namely the lambda calculus, is used. However, it was pointed out by a number of authors, including Joseph Goguen, that first-order equational logic is perfectly adequate for the task, and has some specific advantages.

Algebraic Semantics of Sequential Languages

The choice of first-order equational logic leads to a form of algebraic semantics of sequential imperative languages in which:

- the semantics of a programming language $\mathcal{L}$ is axiomatized as an equational theory $T_{\mathcal{L}}$;
- the mathematical semantics of the language is given by the initial algebra $T_{T_{\mathcal{L}}}$;
- if the equations in $T_{\mathcal{L}}$ are confluent, this also gives an operational semantics to the language, expressed in terms of equational simplification.

Algebraic Semantics of Sequential Languages (II)

In this setting, the program correctness question can be formulated as follows: given a program P in a sequential imperative language $\mathcal{L}$, and given some properties Q about P (where Q typically involves the text of P) we say that P satisfies Q iff,

$$T_{T_{\mathcal{L}}} \models Q.$$

Proof-theoretically, we use an inductive inference system, to try to prove,

$$T_{\mathcal{L}} \vdash_{ind} Q.$$

The Syntax of Java: Types

We illustrate these ideas with a fragment of Java. This fragment includes arithmetic expressions, assignments, sequential composition and loops.

We start with defining types for this language.

```
fmod TYPE is
  sort Type .
  sort TypeWithOutArr .
  subsort TypeWithOutArr < Type .
  ops int boolean String : -> TypeWithOutArr .
endfm
```

The Syntax of Java: Names

Next we introduce names which will be the basis of variables. Please note that the `equalName` operator is not part of the actual Java syntax but rather an extra operator.

```
fmod NAME is ex QID .
  pr INT .
  pr STRING .
  pr BOOL .
  sorts Name NameList .
  subsort Qid < Name < NameList .
  op _[] : Name -> Name [prec 50] .
  op `(`) : -> NameList .
  op _,_ : NameList NameList -> NameList [assoc id: ()] .

  op equalName : Name Name -> Bool .
endfm
```

The Syntax of Java: Variables

Variables are defined this way:

```
fmod VAR-SYNTAX is ex NAME .
  sorts Var Vars .
  subsort Name < Var < Vars .
  subsort NameList < Vars .
  op _,_ : Vars Vars -> Vars [ditto] .
endfm
```

The Syntax of Java: Generic Expressions

This is our definition of generic expressions. Note the lists of expressions.

```
fmod GENERIC-EXP-SYNTAX is ex VAR-SYNTAX .
  sorts Exp Exps StExp .
  subsort Var StExp < Exp < Exps .
  subsort Vars < Exps .
  op #i : Int -> Exp .
  op #s : String -> Exp .
  op #b : Bool -> Exp .
  op _,_ : Exps Exps -> Exps [ditto] .
endfm
```

The Syntax of Java: Arithmetic Expressions

The arithmetic expressions are defined in the following way. In Java they are left-associative which is what the gather pattern assures.

```
fmod ARITH-EXP-SYNTAX is ex GENERIC-EXP-SYNTAX .
  ops _+_ _-_ : Exp Exp -> Exp [prec 40 gather(E e)] .
  ops _*_ _/_ _%_ : Exp Exp -> Exp [prec 35 gather(E e)] .
  op -_ : Exp -> Exp [prec 33] .
endfm
```

The Syntax of Java: Examples

Program variables, of sort `Var`, are terms of the form `'Q` with `Q` a character string such as `"foo"`. They are, of course, completely different from the mathematical variables that we use in equations.

Arithmetic expressions, of sort `Exp`, are then expressions formed from variables and integer constants using the usual arithmetic operations. For example,

```
'X

#i(99)

'X + (#i(7) * 'Y)

'Z - (- 'X + (#i(12) * 'W))
```

The Syntax of Java: Comparison and Bool Expressions

This defines the syntax of comparisons and boolean connectives.

```
fmod REXP-SYNTAX is ex GENERIC-EXP-SYNTAX .
  ops _==_ _!=_ _<_  _<=_ _>_ _>=_ : Exp Exp -> Exp [prec 45] .
endfm

fmod BEXP-SYNTAX is ex GENERIC-EXP-SYNTAX .
  op !_ : Exp -> Exp [prec 47] .
  op _&&_ : Exp Exp -> Exp [prec 49] .
  op _||_ : Exp Exp -> Exp [prec 51] .
endfm
```

For example, `'X == #i(7)` is using a comparison and a boolean connective is used in `'X == #i(7) && B`. Both are of sort `Exp`.

The Syntax of Java: Array and New

Arrays are composed in this way and the syntax of new is straightforward.

```
fmod ARRAY-SYNTAX is ex TYPE .
  ex GENERIC-EXP-SYNTAX .
  op _[] : Type -> Type [prec 20] .
  op _[_] : Type Exp -> Type [prec 20] .
  op _[_] : Exp Exp -> Var [prec 20] .
endfm

fmod NEW-SYNTAX is ex ARRAY-SYNTAX .
  ex GENERIC-EXP-SYNTAX .
  op new_ : Type -> Exp [prec 25 gather(E)] .
endfm
```

As an example, `int[]` and `int[#i(5)]` are of the type integer array, and `a[b]` is the variable at position `b` in the array `a`. A new integer array of length 5 is defined by `new int[#i(5)]`.

The Syntax of Java: Assignment

Next let us present assignments.

```
fmod ASSIGNM-EXP-SYNTAX is ex GENERIC-EXP-SYNTAX .
  op _=_ : Var Exp -> StExp [prec 60] .
endfm
```

An example `StExp` is `'X = 'Y + #i(7)`.

The Syntax of Java: Declaration

This leads us to declarations.

```
fmod DECLARATION-SYNTAX is pr TYPE .
  ex ASSIGNM-EXP-SYNTAX .
  sort Declaration Declarators .
  subsort Name < Declarators .
  subsort StExp < Declarators .
  op __ : Type Declarators -> Declaration [prec 70] .
endfm
```

A `Declarator` is for example `'X` or `'X = #i(0)` and so a `Declaration` would be for example `int 'X = #i(0)`.

The Syntax of Java: Blocks

Next let us present blocks.

```
fmod BLOCK-SYNTAX is
  sorts Block BlockStatements .
  subsort Block < BlockStatements .
  op ; : -> Block .
  op {_} : BlockStatements -> Block .
  op __ : BlockStatements BlockStatements -> BlockStatements
    [assoc prec 125] .
endfm
```

Any code (i.e. any `BlockStatements`) can be put into a `Block` by means of the {_} operator.

The Syntax of Java: Statements

Let us now look at statements:

```
fmod STATEMENT-SYNTAX is ex BLOCK-SYNTAX .
  pr QID .
  sort Statement .
  subsort Block < Statement < BlockStatements .
endfm

fmod EXP-STATEMENT-SYNTAX is ex STATEMENT-SYNTAX .
  ex GENERIC-EXP-SYNTAX .
  op _; : StExp -> Statement [prec 110] .
endfm

fmod DECLARATION-STATEMENT-SYNTAX is ex DECLARATION-SYNTAX .
  ex STATEMENT-SYNTAX .
  op _; : Declaration -> Statement [prec 75] .
endfm
```

A `BlockStatements` example is: `'X = #i(0) ; int 'Y ;.`

The Syntax of Java: Conditional

Here is the definition of conditionals.

```
fmod IF-SYNTAX is ex STATEMENT-SYNTAX .
  ex GENERIC-EXP-SYNTAX .
  op if__else_ : Exp Statement Statement -> Statement [prec 110] .
  op if__ : Exp Statement -> Statement [prec 115] .
endfm
```

An example for a `Statement` of this type is:

```
if ('X <= 'Y) 'X = #i(0) ;
              else {'X = 'Y - #i(1) ; Y = Y + #i(1) ;}
```

The Syntax of Java: Loops

Now we want to show the while and do while loops.

```
fmod WHILE-SYNTAX is ex STATEMENT-SYNTAX .
  ex GENERIC-EXP-SYNTAX .
  op while__ : Exp Statement -> Statement [prec 110] .
endfm
```

```
fmod DO-SYNTAX is ex STATEMENT-SYNTAX .
  ex GENERIC-EXP-SYNTAX .
  op do_while_; : Statement Exp -> Statement [prec 110] .
endfm
```

A loop example is:

```
while ('X < #i(10)) 'X = 'X + #i(1) ;
```

The Syntax of Java: Loops (II)

There are also for loops in our language.

```
fmod FOR-SYNTAX is ex STATEMENT-SYNTAX .
  ex GENERIC-EXP-SYNTAX .
  ex DECLARATION-SYNTAX .
  op for(_;_;_)_ : Exps Exp Exps Statement -> Statement [prec 110] .
  op for(_;_;_)_ : Declaration Exp Exps Statement -> Statement [prec 110
endfm
```

The Syntax of Java

This puts all parts of the syntax together:

```
fmod JAVA-SYNTAX is
  ex ARITH-EXP-SYNTAX .
  ex REXP-SYNTAX .
  ex BEXP-SYNTAX .
  ex ARRAY-SYNTAX .
  ex NEW-SYNTAX .
  ex ASSIGNM-EXP-SYNTAX .
  ex DECLARATION-SYNTAX .
  ex DECLARATION-STATEMENT-SYNTAX .
  ex EXP-STATEMENT-SYNTAX .
  ex IF-SYNTAX .
  ex WHILE-SYNTAX .
  ex DO-SYNTAX .
  ex FOR-SYNTAX .
  sort Pgm .
  subsort BlockStatements < Pgm .
endfm
```

The Syntax of Java: Parsing examples

We can now take a look at a few examples and let Maude parse them using the syntax we defined.

```
parse int 'x = #i(5) ; int 'y = #i(4) ; {'x = #i(42) ; } 'x = 'x + 'y ; .

parse int 'x = #i(5) ; int 'y = #i(4) ; {int 'x = #i(42) ; } 'x = 'x + 'y ;  .

parse int 'x = #i(0) ; if #b(false)
                          if #b(true) 'x = #i(1) ;
                        else 'x = #i(2) ;  .

parse int 'a = #i(3) ; int 'b = #i(2) ; int 'c = #i(1) ;
        int 'd = 'a - 'b + 'c ; .

parse int 'x = #i(7) ; int 'y = #i(5) ; int 't = 'x ; 'x = 'y ; 'y = 't ; .

parse int 'n = #i(5) ; int 'c = #i(0) ; int 'x = #i(1) ;
                  while ('c < 'n) { 'c = 'c + #i(1) ; 'x = 'x * 'c ; } .
```

The Infrastructure for Java: State Infrastructure

So far, we have only described the syntax of our language. To describe its semantics we must specify how the execution of such programs affects the state infrastructure containing the values for the program variables (among other information).

The state infrastructure is defined by the following modules, where we separately present the locations, environments, values, stores and continuations that make up the state.

The Infrastructure for Java: Location

A program variable will not be directly mapped to its value but to a location in the store. This leads to a two level mapping of variables to locations on the one hand and locations to values on the other. This just shows what a location is, an example location is `l(17)`.

```
fmod LOCATION is
  protecting INT .
  sorts Location LocationList .
  subsort Location < LocationList .
  op noLoc : -> LocationList .
  op _,_ : LocationList LocationList -> LocationList [assoc id: noLoc] .
  op l : Nat -> Location .
endfm
```

The Infrastructure for Java: Environment

This module defines what an environment is and also gives equations on how to change it.

```
fmod ENVIRONMENT is protecting LOCATION .
  protecting NAME .
  sort Env .
  op noEnv : -> Env .
  op [_,_] : Name Location -> Env .
  op __ : Env Env -> Env [assoc comm id: noEnv] .

  vars X Y : Name .  vars Env : Env .  vars L L' : Location .
  var Xl : NameList .  var Ll : LocationList .
  op _[_<-_] : Env NameList LocationList -> Env .
  op _[_<-_] : Env Name Location -> Env .
  eq Env[() <- noLoc] = Env .
  eq Env[X,Y,Xl <- L,L',Ll] = (Env [X <- L]) [Y,Xl <- L',Ll] .
```

The Infrastructure for Java: Environment (II)

```
  eq ([X,L] Env)[X <- L'] = ([X,L'] Env) .
  ceq ([Y, L] Env)[X <- L'] = [Y, L] (Env [X <- L'])
    if equalName(Y, X) = false .
  eq noEnv [X <- L'] = [X,L'] .
endfm
```

The environment `(['X, 1(1)] ['Y, 1(2)])['X,'Y,'Z <- 1(3),1(4),1(5)]` evaluates thus to `['X,1(3)] ['Y,1(4)] ['Z,1(5)]`.

The Infrastructure for Java: Value and Store

Values and stores are defined like this. No equations are given for the store though (unlike for the environment).

```
fmod VALUE is
  sorts Value ValueList .
  subsort Value < ValueList .
  op noVal : -> ValueList .
  op _,_ : ValueList ValueList -> ValueList [assoc id: noVal] .
endfm

fmod STORE is protecting LOCATION .
  extending VALUE .
  sort Store .
  op noStore : -> Store .
  op [_,_] : Location Value -> Store .
  op __ : Store Store -> Store [assoc comm id: noStore] .
endfm
```

The Infrastructure for Java: Environments and Stores

The environments and stores are defined in a pretty concrete way for this language. Using a more abstract environment/store concept would have its advantages from the point of view of program verification.

A more abstract concept of environment/stores does not work nicely with the side-effects and hiding that are possible in our language though, so we settled with the concrete variant. This is also done so we can extend this subset of Java to a more complete version of Java in the future.

The Infrastructure for Java: Continuations

Within continuations all the execution context is stored. This can be viewed as "the rest of the program" which needs to be executed. The two operators shown here are the ending point of an execution. Within the semantics we will define other operators with co-domain `Continuation` as needed.

```
fmod CONTINUATION is
  sort Continuation .
  op stop : -> Continuation .
  op res : -> Continuation .
endfm
```

The Infrastructure for Java: Arrays

The arrays are represented this way. The array environment maps indices (of sort `Int`) to locations. An array is built as `a(Type, [0, Loc])` for example.

```
fmod ARRAY-ENV is ex INT . ex LOCATION . pr TYPE .
  sort ArrayEnv .
  op noArrayEnv : -> ArrayEnv .
  op [_,_] : Int Location -> ArrayEnv .
  op __ : ArrayEnv ArrayEnv -> ArrayEnv [assoc comm id: noArrayEnv] .
endfm

fmod ARRAY is ex ARRAY-ENV . ex VALUE .
  sort Array .
  subsort Array < Value .
  op a : Type ArrayEnv -> Array .
endfm
```

The Infrastructure for Java: Outputs

The outputs of our language are values, wrapped in specific operators and connected as below:

```
fmod OUTPUT is ex VALUE .
  ex STORE .
  sort Output .
  op O : Value -> Output .
  op O : Value Store -> Output .
  op noOutput : -> Output .
  op _,_ : Output Output -> Output [assoc id: noOutput] .
endfm
```

The Infrastructure for Java: State

The state is made up of state attributes which are the environment, store, output and a counter for the next free memory location each wrapped in some operator. It has the usual multiset union composition operator.

```
fmod STATE is extending ENVIRONMENT . extending STORE .
  extending CONTINUATION . ex OUTPUT .
  sorts StateAttribute MyState .
  subsort StateAttribute < MyState .
  op empty : -> MyState .
  op _,_ : MyState MyState -> MyState [assoc comm id: empty] .
  op e : Env -> StateAttribute .
  op n : Nat -> StateAttribute .
  op m : Store -> StateAttribute .
  op out : Output -> StateAttribute .
```

The Infrastructure for Java: State (II)

```
  sort SuperState .
  sort WrappedState .
  subsort WrappedState < SuperState .
  op noState : -> WrappedState .
  op state : MyState -> WrappedState .
  op k : Continuation -> SuperState .
  op _,_ : WrappedState WrappedState -> WrappedState
      [assoc comm id: noState] .
  op _,_ : SuperState SuperState -> SuperState
      [assoc comm id: noState] .
endfm
```

This is some more of the state which is needed because we did not want the context, i.e. the `Continuation`, to be part of the state but to be composable with it. So instead of having `e(..), m(..), n(..), out(..), k(..)` we now have `state(e(..), m(..), n(..), out(..)), k(..)` which is of sort `SuperState`.

The Infrastructure for Java: Helpers

There are a few helper functionalities which are needed that are not part of Java but which we use in our evaluation of Java code. This operator returns the value of a variable in a given state.

```
var MYS : MyState . var V : Value . var X : Name .
var L : Location . var Env : Env . var M : Store .
op _[_] : WrappedState Name -> Value .
eq state((MYS, e([X,L] Env), m([L,V] M)))[X]
   = V .
```

The Infrastructure for Java: Helpers (II)

These equations are needed to get non-ground term reduced in the decision procedure we use for proving properties of programs.

```
vars I J I' : Int .
eq 0 + I = I .
eq _-_(I, J) = I + (- J) .
eq I' * (I + J) = (I' * I) + (I' * J) .
```

The Infrastructure for Java: Helpers (III)

These operators extract the initial, guard and body parts of a loop respectively.

```
op extInit_ : BlockStatements -> BlockStatements .
op extGuard_ : BlockStatements -> Exp .
op extBody_ : BlockStatements -> BlockStatements .

var BS : BlockStatements . var E : Exp . var S : Statement .

eq extInit(while E S) = ; .
eq extInit(BS while E S) = BS .
eq extBody(while E S) = S .
eq extBody(BS while E S) = S .
eq extGuard(while E S) = E .
eq extGuard(BS while E S) = E .
```

The Semantics of Java: Generic Expressions

The first two operators allow for expression and value lists on the top of a continuation. The `int` operator creates a value of an integer. The semantics of an integer expression is to become an integer value. This module is the foundation upon which all the following semantic equations rely. Most equations here transform non-Java syntax intermediate continuations which are created by the rules for each feature.

```
fmod GENERIC-EXP-SEMANTICS is protecting GENERIC-EXP-SYNTAX .
  protecting STATE .
  op _->_ : Exps Continuation -> Continuation .
  op _->_ : ValueList Continuation -> Continuation .
  op int : Int -> Value .
  op str : String -> Value .
```

The Semantics of Java: Generic Expressions (II)

Integer and string expressions become integer, respectively string, values.

```
vars E E' : Exp .  var El : Exps .
vars V V' : Value .  var Vl Vl' : ValueList .
var L : Location .  var Ll : LocationList .
var I : Int .  var K : Continuation .
var M : Store .  vars Env Env' : Env .
var str : String . var MYS : MyState .

eq k(#i(I) -> K) = k(int(I) -> K) .
eq k(#s(str) -> K) = k(str(str) -> K) .
```

The Semantics of Java: Generic Expressions (III)

This makes sure that a list of expressions `E1, E2, E3, ...` is evaluated in the given order and has the result `V1, V2, V3, ...` where `Vi` is the result of `Ei`'s evaluation. This is useful to have arguments of a function evaluated. We also always focus only on the first elements of the continuation which is enough because of the way we defined it.

```
op [_|_] -> _ : Exps ValueList Continuation -> Continuation .
eq k((E,E',El) -> K) = k(E -> [E',El | noVal] -> K) .
eq k(V -> [() | Vl] -> K) = k((Vl,V) -> K) .
eq k(V -> [E,El | Vl] -> K) = k(E -> [El | Vl,V] -> K) .
eq k(() -> K) = k(noVal -> K) .
```

The Semantics of Java: Generic Expressions (IV)

Lists of locations can also be on the top of a continuation, as well as the `change` operator which will change the value at the given location to the given value. This will be helpful to define assignments when no result is needed.

```
op _->_ : LocationList Continuation -> Continuation .
op change (_,_) -> _ : Value Location Continuation -> Continuation .
eq k(noVal -> noLoc -> K), state(m(M), MYS) = k(K), state(m(M), MYS) .
eq k((V, Vl) -> (L, Ll) -> K), state(m([L, V'] M), MYS)
     = k(change(V, L) -> (Vl -> Ll -> K)), state(m([L, V'] M), MYS) .

eq k(change(V, L) -> K), state(m([L, V'] M), MYS)
     = k(K), state(m([L, V] M), MYS) .
```

The Semantics of Java: Generic Expressions (V)

Here `set&fetch` creates a second copy of the values that are assigned. Once all the values of the value list are assigned to the locations of the location list the only thing left (outside of the rest `K`) will be exactly the copy of the value list. This allows assignments where the result is also directly used afterwards.

```
op set&fetch_->_ : LocationList Continuation -> Continuation .

eq k(Vl -> set&fetch(Ll) -> K) = k(Vl -> Ll -> Vl -> K) .
```

The Semantics of Java: Generic Expressions (VI)

Restoring environments is important. It can be done while keeping some values on top of the continuation or without such values. This will be used for the handling of blocks.

```
  op _->_ : Env Continuation -> Continuation .
  eq k(Vl -> Env -> K), state(e(Env'), MYS)
   = k(Vl -> K), state(e(Env), MYS) .
  eq k(Env -> K), state(e(Env'), MYS)
   = k(K), state(e(Env), MYS) .
endfm
```

The Semantics of Java: Locations

This gives the semantics for locations on the continuation and will be used to get the location of a variable.

```
fmod LOCATION-SEMANTICS is ex GENERIC-EXP-SEMANTICS . pr ARRAY-SYNTAX .
  pr ARRAY .
  op loc_ -> _ : Exp Continuation -> Continuation .
  op loc -> _ : Continuation -> Continuation .
  vars E E' : Exp . var L : Location . var X : Name .
  var I : Int .  var K : Continuation . var Env : Env .
  var T : Type . var aEnv : ArrayEnv . var MYS : MyState .
  eq k(loc(X) -> K), state(e([X, L] Env), MYS)
       = k(L -> K), state(e([X, L] Env), MYS) .
  eq k(loc(E [ E' ]) -> K) = k((E, E') -> loc -> K) .
  eq k((a(T, [I,L] aEnv), int(I)) -> loc -> K) = k(L -> K) .
endfm
```

The Semantics of Java: Arithmetic Expressions

This gives the semantics for arithmetic expressions. Take for example addition, here first the two arguments are evaluated and then, if they are both integers, they are added together. All other operators are defined similarly.

```
fmod ARITH-EXP-SEMANTICS is protecting ARITH-EXP-SYNTAX .
  extending GENERIC-EXP-SEMANTICS .
  op + -> _ : Continuation -> Continuation .
  op - -> _ : Continuation -> Continuation .
  op * -> _ : Continuation -> Continuation .
  op / -> _ : Continuation -> Continuation .
  op % -> _ : Continuation -> Continuation .
  vars E E' : Exp .  var K : Continuation .  vars I I' : Int .
  vars str str' : String .
  eq k((E + E') -> K) = k((E,E') -> + -> K) .
  eq k((int(I), int(I')) -> + -> K) = k(int(I + I') -> K) .
```

The Semantics of Java: Arithmetic Expressions (II)

```
  eq k((E - E') -> K) = k((E,E') -> - -> K) .
  eq k((int(I), int(I')) -> - -> K) = k(int(I - I') -> K) .
  eq k((- E) -> K) = k(E -> - -> K) .
  eq k(int(I) -> - -> K) = k(int(- I) -> K) .
  eq k((E * E') -> K) = k((E,E') -> * -> K) .
  eq k((int(I), int(I')) -> * -> K) = k(int(I * I') -> K) .
  eq k((E / E') -> K) = k((E,E') -> / -> K) .
  eq k((int(I), int(I')) -> / -> K) = k(int(I quo I') -> K) .
  eq k((E % E') -> K) = k((E,E') -> % -> K) .
  eq k((int(I), int(I')) -> % -> K) = k(int(I rem I') -> K) .
endfm
```

The Semantics of Java: Comparisons

The semantics of comparisons are defined similarly to those of arithmetic expressions. First evaluate the subexpressions and then compare the results. They are all defined similarly.

```
fmod REXP-SEMANTICS is pr REXP-SYNTAX .
  extending GENERIC-EXP-SEMANTICS .
  op bool : Bool -> Value .
  op == -> _ : Continuation -> Continuation .
  op != -> _ : Continuation -> Continuation .
  op < -> _  : Continuation -> Continuation .
  op <= -> _ : Continuation -> Continuation .
  op > -> _  : Continuation -> Continuation .
  op >= -> _ : Continuation -> Continuation .
  vars E E' : Exp .  var K : Continuation .
  vars I I' : Int .  vars B B' : Bool .
  eq k(#b(B) -> K) = k(bool(B) -> K) .
```

The Semantics of Java: Comparisons (II)

```
  eq k((E == E') -> K) = k((E,E') -> == -> K) .
  eq k((int(I),int(I')) -> == -> K) = k(bool(I == I') -> K) .
  eq k((E != E') -> K) = k((E,E') -> != -> K) .
  eq k((int(I),int(I')) -> != -> K) = k(bool(not (I == I')) -> K) .
  eq k((E >= E') -> K) = k((E,E') -> >= -> K) .
  eq k((int(I),int(I')) -> >= -> K) = k(bool(I >= I') -> K) .
  eq k((E > E') -> K) = k((E,E') -> > -> K) .
  eq k((int(I),int(I')) -> > -> K) = k(bool(I > I') -> K) .
  eq k((E <= E') -> K) = k((E,E') -> <= -> K) .
  eq k((int(I),int(I')) -> <= -> K) = k(bool(I <= I') -> K) .
  eq k((E < E') -> K) = k((E,E') -> < -> K) .
  eq k((int(I),int(I')) -> < -> K) = k(bool(I < I') -> K) .
endfm
```

The Semantics of Java: Boolean Expressions

The evaluation of boolean expressions is a bit more complicated, the simple case, negation, can be seen here, conjunction and disjunction follow.

```
fmod BEXP-SEMANTICS is pr BEXP-SYNTAX . ex REXP-SEMANTICS .
  op ! -> _  : Continuation -> Continuation .
  op && -> _ : Continuation -> Continuation .
  op || -> _ : Continuation -> Continuation .

  vars E E' : Exp .  var K : Continuation .
  vars I I' : Int .  vars B B' : Bool .

  eq k((! E) -> K) = k(E -> ! -> K) .
  eq k(bool(B) -> ! -> K) = k(bool(not B) -> K) .
```

The Semantics of Java: Boolean Expressions (II)

The evaluation of boolean expressions needs a bit more subtlety. According to the Java Language Specification the result of a conjunction is `false` whenever the first argument evaluates to `false` and in that case the second argument is not evaluated! The definition for disjunction works similarly.

```
  eq k((E && E') -> K) = k(E -> && -> (E' -> K)) .
  eq k(bool(false) -> && -> (E' -> K)) = k(bool(false) -> K) .
  eq k(bool(true) -> && -> (E' -> K)) = k(E' -> && -> (bool(true) -> K)) .
  eq k(bool(B) -> && -> (bool(B') -> K)) = k(bool(B' and B) -> K) .

  eq k((E || E') -> K) = k(E -> || -> (E' -> K)) .
  eq k(bool(true) -> || -> (E' -> K)) = k(bool(true) -> K) .
  eq k(bool(false) -> || -> (E' -> K)) = k(E' -> || -> (bool(false) -> K)) .
  eq k(bool(B) -> || -> (bool(B') -> K)) = k(bool(B' or B) -> K) .
endfm
```

The Semantics of Java: Assignment

The definition of assignment makes use of the previously defined `set&fetch` and `loc` operators and is therefore straightforward.

```
fmod ASSIGNM-EXP-SEMANTICS is ex ASSIGNM-EXP-SYNTAX .
  ex GENERIC-EXP-SEMANTICS .
  ex LOCATION-SEMANTICS .
  pr ARITH-EXP-SEMANTICS .
  op =_ -> _ : Exp Continuation -> Continuation .
  var E : Exp .  var K : Continuation .
  var L : Location .
  var X : Var .
  eq k((X = E) -> K) = k(loc(X) -> = (E) -> K) .
  eq k(L -> = (E) -> K) = k(E -> set&fetch(L) -> K) .
endfm
```

The Semantics of Java: Arrays

Arrays are accessed in the obvious way. Giving an integer as an argument to the array, if that index is defined then the corresponding memory location is accessed and the value returned.

```
fmod ARRAY-SEMANTICS is ex ARRAY-SYNTAX  .
  ex GENERIC-EXP-SEMANTICS .
  ex ARRAY .
  op [] -> _ : Continuation -> Continuation .
  vars E E' : Exp . var T : Type . var aEnv : ArrayEnv . var M : Store .
  var I : Int . var L : Location . var V : Value . var K : Continuation .
  var MYS : MyState .
  eq k((E [ E' ]) -> K) = k((E, E') -> [] -> K) .
  eq k((a(T, [I,L] aEnv), int(I)) -> [] -> K), state(m([L,V] M), MYS)
       = k(V -> K), state(m([L,V] M), MYS) .
endfm
```

The Semantics of Java: Field Access

In our subset of Java field access boils down to retrieving the value a (local) variable has, as arrays are handled separately. This also uses a previously defined helper operator.

```
fmod FIELD-SEMANTICS is ex GENERIC-EXP-SEMANTICS .
  ex LOCATION-SEMANTICS .
  ex HELPING-OPERATORS .
  var X : Name . var K : Continuation .
  var MYS : MyState .
  eq k(X -> K), state(MYS)  = k((state(MYS)[X]) -> K), state(MYS) .
endfm
```

The Semantics of Java: Types

The types of declarations and values are defined this way. These are also helper operators and not part of the Java syntax.

```
fmod GET-TYPES is ex TYPE . ex GENERIC-EXP-SEMANTICS .
  ex ARRAY-SEMANTICS . ex DECLARATION-SYNTAX .
  op GetType : Declaration -> Type .
  var qid : Qid . vars T : Type . var dn : Name . var E : Exp .
  var I : Int . var str : String . var aEnv : ArrayEnv .
  eq GetType(T qid) = T .
  eq GetType(T dn[]) = GetType((T []) dn) .
  eq GetType(T dn = E) = GetType(T dn) .
  op GetType : Value -> Type .
  eq GetType(int(I)) = int .
  eq GetType(str(str)) = String .
  eq GetType(a(T, aEnv)) = T .
endfm
```

The Semantics of Java: Blocks

The semantics of blocks is that all changes to the environment that happen within the block are discarded at the end of the block. Otherwise the execution proceeds as usual.

```
fmod BLOCK-SEMANTICS is ex BLOCK-SYNTAX .
  ex GENERIC-EXP-SEMANTICS .
  op _ -> _ : BlockStatements Continuation -> Continuation .
  var K : Continuation . var bs : BlockStatements . var Env : Env .
  var MYS : MyState .
  eq k( ; -> K) = k(K) .
  eq k({bs} -> K), state(e(Env), MYS)
       = k(bs -> Env -> K), state(e(Env), MYS) .
endfm
```

The Semantics of Java: Declarations

For declarations we first evaluate the expression that is assigned to the newly declared variable, if there is any. Then the variable is mapped to the next free location in the environment and that location is mapped to the initial value of the appropriate type in the store. If there is an assignment to the variable it is processed next, see next slide.

```
fmod DECLARATION-SEMANTICS is ex DECLARATION-STATEMENT-SYNTAX .
  ex GENERIC-EXP-SEMANTICS . ex GET-TYPES . ex BLOCK-SEMANTICS .
  op _ -> _ : Declaration Continuation -> Continuation .
  var T : Type . var K : Continuation . var dn : Name .
  var E : Exp . var Env : Env . var V : Value . var N : Int .
  var M : Store . var qid : Qid . var dc : Declaration .
  var bs : BlockStatements . var MYS : MyState .
  eq k((dc ;) -> K) = k(dc -> K) .
  eq k(((dc ;) bs) -> K) = k(dc -> bs -> K) .
```

The Semantics of Java: Declarations (II)

For this we also need to be able to get the name out of a declaration and define the initial values of any type, as in Java, unlike C/C++, a '0' value is assigned to each newly declared variable. `GetName` and `alloc` will be defined next.

```
eq k((T dn = E) -> K) = k(E -> (T dn) -> K) .
eq k((T dn) -> K), state(e(Env), n(N), m(M), MYS) =
   k(K), state(e(Env[GetName(dn) <- l(N)]), n(N + 1),
               m(M[l(N), alloc(GetType(T dn))]), MYS) .
eq k(V -> (T dn) -> K), state(e(Env), n(N), m(M), MYS) =
   k(change(V, l(N)) -> K),
   state(e(Env[GetName(dn) <- l(N)]), n(N + 1),
         m(M[l(N), alloc(GetType(T dn))]), MYS) .
```

The Semantics of Java: Declarations (III)

This defines the helpers `GetName` and `alloc`.

```
  op GetName : Name -> Qid .
  eq GetName(qid) = qid .
  eq GetName(dn[]) = GetName(dn) .
  op alloc : Type -> Value .
  eq alloc(int) = int(0) .
  eq alloc(String) = str("") .
  eq alloc(T[]) = a(T[], noArrayEnv) .
endfm
```

The Semantics of Java: Output

Outputs are gathered in the output state attribute in the order in which they are printed.

```
fmod OUTPUT-METHOD is ex GENERIC-EXP-SEMANTICS .
  pr OUTPUT .
  op print -> _ : Continuation -> Continuation .
  var E : Exp . var V : Value . var K : Continuation .
  var O : Output . var MYS : MyState .
  op System.out.print : Exp -> Exp .
  eq k((System.out.print(E)) -> K) = k(E -> print -> K) .
  eq k(V -> print -> K), state(out(O), MYS)
       = k(K), state(out(O,O(V)), MYS) .
endfm
```

The Semantics of Java: New

We only have arrays to worry about for the `new` operator, as we do not use objects. Here we define some helping operators.

```
fmod NEW-SEMANTICS is ex NEW-SYNTAX . pr GENERIC-EXP-SEMANTICS .
  ex GET-TYPES . ex DECLARATION-SEMANTICS . ex BLOCK-SEMANTICS .
  op GetArrayType : Type -> Type .
  op GetExps : Type -> Exps .
  var K : Continuation . var Vl : ValueList . var array : Array .
  var aEnv : ArrayEnv . vars I N : Int . var V : Value .
  var M : Store . var MYS : MyState . var Inotzero : NzInt .
  var TnoArr : TypeWithOutArr . var T : Type . var E : Exp .
  eq GetArrayType(T[E]) = GetArrayType(T) [] .
  eq GetArrayType(T[]) = GetArrayType(T) [] .
  eq GetArrayType(TnoArr) = TnoArr .
  eq GetExps(T[E]) = GetExps(T), E .
  eq GetExps(T[]) = GetExps(T) .
  eq GetExps(TnoArr) = () .
```

The Semantics of Java: New (II)

Now this defines how a new array is created and initialized.

```
  op newArray _ -> _ : Type Continuation -> Continuation .
  op newArray (_,_,_) -> _ : Array ValueList Int Continuation -> Continuation .
  eq k((new T) -> K) = k(GetExps(T) -> newArray (GetArrayType(T)) -> K) .
  eq k((int(I), Vl) -> newArray (T) -> K)
       = k(newArray(a(T, noArrayEnv), Vl, I) -> K) .
  eq k(noVal -> newArray(int) -> K) = k(int(0) -> K) .
  eq k(noVal -> newArray(String) -> K) = k(str("") -> K) .
  eq k(noVal -> newArray(T[]) -> K) = k(a(T[], noArrayEnv) -> K) .
  eq k(newArray(array, Vl, 0) -> K) = k(array -> K) .
  eq k(newArray(a(T[], aEnv), Vl, Inotzero) -> K) =
     k(Vl -> newArray(T) -> (newArray(a(T[], aEnv), Vl, (Inotzero - 1)) -> K)) .
  eq k(V -> newArray(a(T, aEnv), Vl, I) -> K), state(m(M), n(N), MYS) =
     k(newArray(a(T, ([I, l(N)] aEnv)), Vl, I) -> K),
     state(m(M[l(N), V]), n(N + 1), MYS) .
endfm
```

The Semantics of Java: Statements

A list of statements, called `BlockStatements` is split apart into the first statement and the rest of the list of statements. With this and the definition of declarations we never need to consider sequential compositions of statements but can focus on one statement at a time.

```
fmod STATEMENT-SEMANTICS is ex STATEMENT-SYNTAX .
  ex BLOCK-SEMANTICS .
  var st : Statement . var bs : BlockStatements .
  var K : Continuation .
  eq k((st bs) -> K) = k(st -> bs -> K) .
endfm
```

An example of this is `'x = 'x + 'Y ; 'y = 'y * 'y ; -> K` which is transformed to `'x = 'x + 'Y ; -> 'y = 'y * 'y ; -> K`

The Semantics of Java: Statements (II)

An assignment as a statement is executed and afterwards the resulting value, which is the one of the right-hand side expression, is dropped.

```
fmod EXP-STATEMENT-SEMANTICS is ex EXP-STATEMENT-SYNTAX .
  ex GENERIC-EXP-SEMANTICS .
  ex STATEMENT-SEMANTICS .
  var SE : StExp . var K : Continuation . var Vl : ValueList .
  eq k((SE ; ) -> K) = k(SE -> ; -> K) .
  eq k(Vl -> ; -> K) = k(K) .
endfm
```

The Semantics of Java: Conditionals

Conditionals first evaluate their condition and then execute the correct branch.

```
fmod IF-SEMANTICS is ex IF-SYNTAX . ex GENERIC-EXP-SEMANTICS .
  ex STATEMENT-SEMANTICS . ex BEXP-SEMANTICS .
  op ? (_,_) -> _ : Statement Statement Continuation -> Continuation .
  var E : Exp . vars St St' : Statement . var K : Continuation .
  eq k((if E St else St') -> K) = k(E -> ? (St, St') -> K) .
  eq k((if E St) -> K) = k(E -> ? (St, ;) -> K) .
  eq k(bool(true) -> ? (St, St') -> K) = k(St -> K) .
  eq k(bool(false) -> ? (St, St') -> K) = k(St' -> K) .
endfm
```

An example for this is `if #b(true) 'x = 'x + #i(1) ; -> K` becomes `#b(true) -> ? ('x = 'x + #i(1) ;, ;) -> K` which in turn becomes `'x = 'x + #i(1) ; -> K`.

The Semantics of Java: Loops

While loops make use of the conditional to decide whether the loop needs to be executed once more.

```
fmod WHILE-SEMANTICS is ex WHILE-SYNTAX . ex GENERIC-EXP-SEMANTICS .
  ex STATEMENT-SEMANTICS . ex IF-SEMANTICS .
  var E : Exp . var St : Statement . var K : Continuation .
  eq k((while E St) -> K) = k(E -> ?({St while E St}, ;) -> K) .
endfm
```

The Semantics of Java: Loops (II)

The `do_while_` and `for` loops are transformed into `while` loops and handled accordingly.

```
fmod DO-SEMANTICS is ex DO-SYNTAX . ex GENERIC-EXP-SEMANTICS .
  ex STATEMENT-SEMANTICS . ex WHILE-SEMANTICS .
  var E : Exp . var St : Statement . var K : Continuation .
  eq k((do St while E ;) -> K) = k(St -> (while E St) -> K) .
endfm

fmod FOR-SEMANTICS is ex FOR-SYNTAX . ex GENERIC-EXP-SEMANTICS .
  ex DECLARATION-SEMANTICS . ex STATEMENT-SEMANTICS .
  ex WHILE-SEMANTICS . ex EXP-STATEMENT-SYNTAX .
  var E : Exp . vars Se Se' : StExp . var St : Statement .
  var K : Continuation . var dc : Declaration .
  eq k((for (Se ; E ; Se') St) -> K)
      = k((Se ; while (E) { St Se' ; }) -> K) .
  eq k((for (dc ; E ; Se') St) -> K)
      = k((dc ; while (E) { St Se' ; }) -> K) .
endfm
```

The Semantics of Java: Completion

The full Java subset we are defining is constituted by all the given submodules.

```
fmod JAVAX is ex JAVA-SYNTAX .
  ex ARITH-EXP-SEMANTICS . ex REXP-SEMANTICS .
  ex BEXP-SEMANTICS . ex ASSIGNM-EXP-SEMANTICS .
  ex ARRAY-SEMANTICS . ex FIELD-SEMANTICS .
  ex BLOCK-SEMANTICS . ex DECLARATION-SEMANTICS .
  ex NEW-SEMANTICS . ex STATEMENT-SEMANTICS .
  ex EXP-STATEMENT-SEMANTICS . ex IF-SEMANTICS .
  ex WHILE-SEMANTICS . ex DO-SEMANTICS .
  ex FOR-SEMANTICS . ex OUTPUT .
```

The Semantics of Java: Completion (II)

The initial state is defined as a convenience as follows.

```
op initial : -> WrappedState .
eq initial = state(e(noEnv), m(noStore), n(0), out(noOutput)) .

var MYS : MyState . var M : Store . var N : Nat . var X : Name .
var I : Int . var BS : BlockStatements . var E : Exp . var V : Value .
var B : Bool . var Env : Env . var O : Output .
```

The Semantics of Java: Completion (III)

From a state and an expression (`Exp`) or code (`BlockStatements`) the following executable `SuperState` variants are created.

```
op _|_ : WrappedState BlockStatements -> SuperState [gather(E e)] .
op _|_ : WrappedState Exp -> SuperState [gather(E e)] .
eq (state(MYS) | BS)
     = (state(MYS), k(BS -> stop)) .
eq (state(MYS) | E)
     = (state(MYS), k(E -> stop)) .

eq (state(MYS), k(V -> stop))
     = state(MYS) .
eq (state(MYS), k(stop))
     = state(MYS) .
```

The Semantics of Java: Completion (IV)

Evaluating a variable in a given state is done in this way. Boolean expressions can also be evaluated and their result be returned.

```
op evalInt : WrappedState Var -> Int .
eq evalInt(state(MYS), X) = int-val(state(MYS)[X]) .

op evalTst : SuperState -> Bool .
op evalTst : WrappedState Exp -> Bool .
eq evalTst(state(e(Env), m(M), n(N), out(O)), E)
     = evalTst(state(e(Env), m(M), n(N), out(O)), k(E -> res)) .
eq evalTst(state(MYS), k(bool(B) -> res)) = B .
```

The Semantics of Java: Completion (V)

The following is necessary because we need specific starting states for proving properties with the ITP. It creates a starting state with some known variables which are mapped to generic values, representative of whatever is there (only `int` currently).

```
op any-int : Int -> Int .
op any-loc : Int -> Location .
op ctxState : BlockStatements WrappedState -> WrappedState .
op ctxState : BlockStatements Int WrappedState -> WrappedState .
op ctxState : BlockStatements -> WrappedState .
eq ctxState(B:BlockStatements)
     = ctxState(B:BlockStatements,
                state(e(noEnv), n(0), m(noStore), out(noOutput))) .
eq ctxState((B:BlockStatements), state(e(Env), n(N), m(M), out(O)))
     = ctxState((B:BlockStatements), 0,
                 state(e(Env), n(N), m(M), out(O))) .
```

The Semantics of Java: Completion (VI)

This finalizes the starting state creation for use with the ITP.

```
  eq ctxState((int X ; B:BlockStatements), I,
                state(e(Env), n(N), out(O), m(M)))
       = ctxState(B:BlockStatements, I + 1,
           state(e([X, any-loc(I)] Env), n(N), out(O),
                 m([any-loc(I), int(any-int(I))] M))) .

  eq ctxState((int X ;), I, state(e(Env), n(N), m(M), out(O)))
       = state(e([X, any-loc(I)] Env), n(N), out(O),
               m([any-loc(I), int(any-int(I))] M)) .

  op int-val : Value -> Int .
  eq int-val(int(I)) = I .
endfm
```

An Interpreter

The above functional module has given a precise axiomatization of our chosen subset of Java that is sufficient for reasoning and program verification purposes. But it has done more than that. Since the semantic equations are ground confluent, the above equations have also given an operational semantics to this language.

Indeed, we can describe the execution of the language by algebraic simplification with the equations from left to right.

Therefore, the above equations have in essence given us an interpreter for our language.

An Interpreter (II)

As a reminder, you can create an initial state by just writing `initial`, you can then execute some code, say `bs` from that state with `initial | bs`.

Make sure to use parentheses around `bs`, otherwise there will be parsing problems. If you now don't want to see the complete resulting state but only the value of a specific variable `'x` then `(initial | bs)['x]` will do that for you.

Mind that this gives you a value, so if you expect to see a `42` you will get a `int(42)` which is the integer value used by our language.

An Interpreter (III)

Some examples are:

```
red (initial | (int 'x = #i(5) ; int 'y = #i(4) ;
                {'x = #i(42) ; } 'x = 'x + 'y ;))['x] .

red (initial | (int 'x = #i(5) ; int 'y = #i(4) ;
                {int 'x = #i(42) ; } 'x = 'x + 'y ;))['x]  .
```

which return

```
rewrites: 86 in 10ms cpu (10ms real) (8600 rewrites/second)
result Value: int(46)
```

respectively, because of shadowing of the assignment to `'x` in the block,

```
rewrites: 86 in 0ms cpu (0ms real) (~ rewrites/second)
result Value: int(9)
```

An Interpreter (IV)

```
red (initial | (int 'x = #i(0) ; if #b(false)
                          if #b(true) 'x = #i(1) ;
                        else 'x = #i(2) ;))['x]  .
```

This is interesting because it shows us how the dangling else problem is solved in this particular implementation.

According to the Java Language Specification an else block belongs to the innermost if part, unless parentheses show otherwise. The result in this case is:

```
rewrites: 18 in 0ms cpu (0ms real) (~ rewrites/second)
result Value: int(0)
```

It would have been `int(2)` if the else was attributed (wrongly) to the outer if.

An Interpreter (V)

```
red (initial | (int 'a = #i(3) ; int 'b = #i(2) ; int 'c = #i(1) ;
                int 'd = 'a - 'b + 'c ;))['d] .
```

With this example, which returns

```
rewrites: 241 in 0ms cpu (0ms real) (~ rewrites/second)
result Value: int(2)
```

we have shown the left-associativity of arithmetic operators. With right-associativity the result would have been `0`.

An Interpreter (VI)

```
red (initial | (int 'x = #i(7) ; int 'y = #i(5) ;
                int 't = 'x ; 'x = 'y ; 'y = 't ;))['x] .
red (initial | (int 'x = #i(7) ; int 'y = #i(5) ;
                int 't = 'x ; 'x = 'y ; 'y = 't ;))['y] .
```

A simple swap example indeed swaps the values of `'x` and `'y`, the results are `5`, respectively `7`, as expected.

```
red (initial | (int 'n = #i(5) ; int 'c = #i(0) ; int 'x = #i(1) ;
        while ('c < 'n) { 'c = 'c + #i(1) ; 'x = 'x * 'c ; })) ['x] .
```

Finally this computes the factorial of `5` and thus returns:

```
rewrites: 416 in 0ms cpu (0ms real) (~ rewrites/second)
result Value: int(120)
```

Formal Reasoning

In fact, the equational axiomatization of our language is much more than an interpreter.

Since we have axiomatized the language as an equational theory, we can do inference about our programs using such a theory.

In fact, since the semantic equations are ground-confluent, such equational reasoning is mechanically supported by the Maude system. Therefore, we can do a substantial amount of formal reasoning about programs in our language, even without using induction.

Formal Reasoning (II)

We may reason about some properties of a given program. In general, of course, such reasoning may require induction, but in some cases plain equational deduction will deliver the goods.

Consider, for example, the following informal specification of a swap program, swapping the values of, say, variables `'X` and `'Y`, namely, that for any state the values in `'X` and `'Y` should have been swapped after execution.

Formal Reasoning (III)

Actually we cannot do that for any state but only for states in which `'X` and `'Y` are already declared. Such a state is created by `ctxState((int 'X ; 'int 'Y ;))`. The fact that this represents any state with `'X` and `'Y` declared results from the generic values put into both variable's locations.

Whether other variables are declared or not does not matter, as long as a candidate for the swap program does not use other variables without declaring them first. This is verified with a quick look at the semantics.

Formal Reasoning (IV)

This correctness specification can be made mathematically precise as the equations:

```
(ctxState((int 'X ; int 'Y ;)) | (swap))['X]
= (ctxState((int 'X ; int 'Y ;)))['Y]

(ctxState((int 'X ; int 'Y ;)) | (swap))['Y]
= (ctxState((int 'X ; int 'Y ;)))['X]
```

Consider now a possible candidate for our `swap` program, namely the program,

```
int 'T = 'X ; 'X = 'Y ; 'Y = 'T ;
```

Formal Reasoning (V)

We can verify that this program meets its specification just by equational simplification as follows,

```
reduce in JAVAX :
ctxState((int 'X ; int 'Y ;)) | (int 'T = 'X ; 'X = 'Y ; 'Y = 'T ;)['X]
== ctxState((int 'X ; int 'Y ;))['Y] .
rewrites: 112 in 0ms cpu (0ms real) (~ rewrites/second)
result Bool: true
==========================================
reduce in JAVAX :
ctxState((int 'X ; int 'Y ;)) | (int 'T = 'X ; 'X = 'Y ; 'Y = 'T ;)['Y]
== ctxState((int 'X ; int 'Y ;))['X] .
rewrites: 112 in 0ms cpu (0ms real) (~ rewrites/second)
result Bool: true
```

Preconditions

Recall the specification of correctness for the program `swap` to swap the value of variable `'X` and variable `'Y` that we have already discussed. With S taking the place of `ctxState((int 'X ; int 'Y ;))`, but with the universal quantification implicitly limited to states of that form we get the specification,

$$(\forall \texttt{I} : \texttt{Int})(\forall \texttt{J} : \texttt{Int})(\forall \texttt{S} : \texttt{State})\texttt{(S)['Y]} = \texttt{int(I)} \wedge \texttt{(S)['X]} = \texttt{int(J)}$$

$$\Rightarrow \texttt{(S | (swap))['X]} = \texttt{int(I)} \wedge \texttt{(S | (swap))['Y]} = \texttt{int(J)}$$

Here we have the implicit precondition that we are starting in a state where `'X` and `'Y` are declared as described above.

Preconditions (II)

We shall call the equation

$$\texttt{(S)['Y]} = \texttt{int(I)} \wedge \texttt{(S)['X]} = \texttt{int(J)}$$

which is assumed to hold before the execution of the program, the precondition.

Note that this equation has a single occurrence of the state variable `S` in each equation, and can be thought of as a state predicate, having the integer variables `I` and `J` as parameters.

Postconditions

Consider in the above specification the equation

$$\texttt{(S | (swap))['X]} = \texttt{int(I)} \wedge \texttt{(S | (swap))['Y]} = \texttt{int(J)}$$

which is supposed to hold after the execution of a program. This can also be viewed as a state predicate, namely the state predicate

$$(\dagger) \quad \texttt{(S)['X]} = \texttt{int(I)} \wedge \texttt{(S)['Y]} = \texttt{int(J)}$$

applied not to `S`, but instead to the state `S | (swap)` after the execution. We call (†) the postcondition. Note that it also has the integer variables `I` and `J` as extra parameters.

State Predicates

This example suggests a general notion of "state predicate," intuitively a property that holds or doesn't hold of a state, perhaps relative to some extra data parameters. Since in our language the only data are integers, such parameters must be integer variables.

Therefore, for our language we can define a state predicate as a conjunction of equations

$$t_1 = t'_1 \ \wedge \ldots \wedge t_n = t'_n$$

in a module extending `JAVAX`, such that the set V of variables in all terms in the equations has at most one variable `S` of sort `State`, which possibly appears more than once, and the remaining variables are all of sort `Int`.

State Predicates (II)

How general is this definition of state predicate?

One can of course generalize things further, by allowing an arbitrary first-order formula (with the same condition on its variables V) instead of just a conjuction of equations. Also, the notion extends naturally to other sequential languages which may have other data structures besides integers.

However, in practice the notion is quite general; among other things because, using an equationally defined equality predicate, we can express arbitrary Boolean combinations of equations as a single equation.

State Predicates (III)

Note that, although a state predicate has at most one variable, say `S`, of sort `State`, it is perfectly possible for `S` to be mentioned more than once. For example,

```
S['X] = S['Y]
```

is a perfectly acceptable state predicate; and `S` could likewise appear in several conjuncts of a state predicate.

Hoare Triples

The above example of our specification for `swap` is paradigmatic of a general way of specifying properties of a sequential imperative program p by means of a Hoare triple (after C.A.R. Hoare)

$$\{A\}\ p\ \{B\}$$

where A and B are state predicates, called, respectively, the precondition, and postcondition of the triple.

In this notation, the specification of `swap` becomes rephrased as,

$$\{\texttt{(S)['Y]} = \texttt{int(I)} \wedge \texttt{(S)['X]} = \texttt{int(J)}\}$$

$$\texttt{swap}$$

$$\{\texttt{(S)['X]} = \texttt{int(I)} \wedge \texttt{(S)['Y]} = \texttt{int(J)}\}$$

Hoare Triples (II)

Given our algebraic approach to the semantics of imperative programs, this is all just an (indeed very useful) *façon de parler* about an ordinary first-order property satisfied by the initial model of our language, namely the initial algebra $T_{\texttt{JAVAX}}$.

Therefore, we define the partial correctness of a program `p` with respect to a Hoare triple by the equivalence,

$$T_{\texttt{JAVAX}} \models \{A\}\ \texttt{p}\ \{B\} \quad \Leftrightarrow$$

$$T_{\texttt{JAVAX}} \models (\forall V)\ A \wedge\ (\texttt{(S | p)} : \texttt{WrappedState}) \Rightarrow (B(\texttt{S/S | p})).$$

Note that `p` terminates iff `(S | p) : WrappedState`.

Hoare Triples (III)

In our `swap` example this becomes,

$$T_{\texttt{JAVAX}} \models (\forall \texttt{I} : \texttt{Int})(\forall \texttt{J} : \texttt{Int})(\forall \texttt{S} : \texttt{State})\,\texttt{(S)['Y]} = \texttt{int(I)} \wedge \texttt{(S)['X]} = \texttt{int(J)}$$

$$\wedge\ \texttt{(S | swap) : WrappedState}$$

$$\Rightarrow \texttt{(S | (swap))['X]} = \texttt{int(I)} \wedge \texttt{(S | (swap))['Y]} = \texttt{int(J)}.$$

which is just our original correctness condition with the addition of the termination condition `(S | swap) : WrappedState`. What we mean by partial correctness, as opposed to total correctness is that termination, rather than being always required, becomes an additional assumption needed for the postcondition to hold.

Of course, since `swap` was a terminating program, this was superfluous in that case, but it is not superfluous when iterations are involved.

Compositionality: Hoare Logic

An important contribution of Hoare was to propose his triples as a compositional logic of programs, by giving a collection of inference rules based on the structure of the program text to decompose proof of correctness of more complex programs into proofs for simpler subprograms.

For example, to prove the correctness of a sequential composition `p q` he gave the rule,

$$\frac{\{A\}\ \texttt{p}\ \{B\} \qquad \{B\}\ \texttt{q}\ \{C\}}{\{A\}\ \ \texttt{p}\ \ \texttt{q}\ \ \{C\}}$$

which can be easily justified by analyzing the semantics of the triples.

Compositionality: Hoare Logic (II)

Also a very useful rule, of easy justification, is the consequence rule,

$$\frac{A \Rightarrow A_1 \quad \{A_1\}\ \mathtt{p}\ \{B_1\} \quad B_1 \Rightarrow B}{\{A\}\ \mathtt{p}\ \{B\}}$$

Another rule of easy justification is the rule for the skip program `;`, that, rephrased in our terms, takes the form,

$$\frac{}{\{A\}\ ;\ \{A\}}.$$

Compositionality: Hoare Logic (III)

All the Hoare rules mentioned so far are quite generic, in the sense that they will apply to many languages. But note that there isn't a single Hoare logic: different programming languages may have somewhat different Hoare rules, depending on their semantics.

In our Java fragment the fact that tests may have side effects makes it necessary to give slightly more subtle Hoare rules for conditionals and loops than those originally proposed by Hoare for those two constructs, because Hoare did not contemplate side-effecting tests in his original language.

Compositionality: Hoare Logic (IV)

Indeed, for conditionals we need to work a little harder. Here, `evalTst(S, TE)` gives the boolean which the evaluation of the test expression `TE` in state `S` returns. With `|` we create an operator which separates the "execution" of a test expression from the rest of the program. It is not possible to use the usual concatenation __ because the test expression is not of the same sort, but using the `|` operator it can be evaluated first, so its side effects change the state, and then the result gets thrown away and the execution continues as usual.

$$\frac{\{A \wedge (\texttt{evalTst(S, t)} = \texttt{true})\}\ \texttt{t | p}\ \{B\} \quad \{A \wedge (\texttt{evalTst(S, t)} = \texttt{false})\}\ \texttt{t | q}\ \{B\}}{\{A\}\ \texttt{if t p else q}\ \{B\}}$$

Compositionality: Hoare Logic (V)

The above captures the usual semantics of `if`, just as in simpler languages, but in contrast here we have this `t |` in front of the execution of the two branches of the conditional in the respective cases. It is not enough here to know that `t` evaluated to either true or false, which is what the two properties assure, but also `t` needs to have been executed for its possible side effects to take effect and change the state. This Hoare rule still simplifies things as we now do not have to take a decision based on the test value anymore but we just have to have the test expression executed. One could also give a sequential composition rule for `|` additionally. The extra effort here is necessary because of side effects!

The Loop Rule and Invariants

A very important rule is the proof rule for the partial correctness of while loops. Here we face the same problem as with conditionals because the loop condition can also have side effects. It takes the form,

$$\frac{\{A \wedge (\texttt{evalTst(S,t)} = \texttt{true})\}\ \texttt{t | p}\ \{A\} \quad \{A \wedge (\texttt{evalTst(S,t)} = \texttt{false})\}\ \texttt{t}\ \{A \wedge (\texttt{evalTst(S,t)} = \texttt{false})\}}{\{A\}\ \texttt{while t p}\ \{A \wedge (\texttt{evalTst(S,t)} = \texttt{false})\}}$$

It requires a somewhat more involved justification. The state predicate A is called an invariant of the loop. Note that, by the partial correctness interpretation of Hoare triples, the holding of the postcondition implicitly depends on the termination assumption for the loop.

Factorial Example

Consider again our factorial program. To prove its correctness, intuitively that it correctly computes the factorial function, we first need to define mathematically such a function, by adding to `Int` an error supersort `Int?` of `Int`, a function,

```
op _! : Int -> Int? .
```

and equations,

```
var I : Int .
eq 0 ! = 1 .
ceq I ! = I * (I - 1)! if 0 < I .
```

Of course, the factorial function is not defined for negative numbers, so that a program may not even terminate if the original input is negative.

Factorial Example (II)

Therefore, we should give the requirement that the input variable `'N` is nonnegative as a precondition, yielding the specification,

$\{(\texttt{S['N]} = \texttt{int(I)}) \wedge (\texttt{0 <= I} = \texttt{true})\}$ `factp` $\{\texttt{S['X]} = \texttt{int(I !)}\}$.

The above specification takes the point of view of a customer who specifies properties of the desired program. An implementer may then give to the customer the following `factp` program:

```
'C = #i(0) ; 'X = #i(1) ;
while ('C < 'N) { 'C = 'C + #i(1) ; 'X = 'X * 'C ; }
```

Factorial Example (III)

The question, then, is how to prove this program correct. To do so we can:

- use the Hoare logic rules, which we have justified, and
- use inductive reasoning, since the correctness of Hoare triples reduces to satisfaction of first-order formulas in the initial model $T_{\texttt{JAVAX}}$.

Justification of the Loop Rule

We have justified other rules in Hoare's logic, but we still need a justification for the soundness of the loop rule,

$$\frac{\{A \wedge (\texttt{evalTst(S,t)} = \texttt{true})\}\ \texttt{t | p}\ \{A\} \quad \{A \wedge (\texttt{evalTst(S,t)} = \texttt{false})\}\ \texttt{t}\ \{A \wedge (\texttt{evalTst(S,t)} = \texttt{false})\}}{\{A\}\ \texttt{while t p}\ \{A \wedge (\texttt{evalTst(S,t)} = \texttt{false})\}}$$

Observations and Lemmas

We approach the justification of the loop rule by means of some observations and auxiliary lemmas that will place us in a good position to prove its soundness.

First Observation: All terms of sort `WrappedState` have a canonical form by the equations.

Second Observation: Using the assumption that the semantic rules of `JAVAX` are ground confluent, for any ground terms s of sort `WrappedState` and p of sort `BlockStatements` we have,

$$
\begin{aligned}
& E \vdash (\forall \emptyset)\ \texttt{s | p} : \texttt{WrappedState} \\
\Leftrightarrow \quad & (\exists! s' \in T_{\texttt{JAVAX},\texttt{WrappedState}})\ \texttt{s | p} \xrightarrow{*}_E s',
\end{aligned}
$$

where E are the equations of `JAVAX`.

Observations and Lemmas (II)

Lemma: For any ground terms s_0 of sort `WrappedState`, t of sort `Exp` (and assuming it will evaluate to a boolean value), and p of sort `BlockStatements`, if we have,

$$E \vdash (\forall \emptyset)\ s_0\ \texttt{|}\ \texttt{while}\ t\ p\ :\ \texttt{WrappedState}$$

then any rewriting sequence, $s_0\ \texttt{|}\ \texttt{while}\ t\ p \xrightarrow{*}_E s'$ with $s' \in T_{\texttt{JAVAX},\texttt{WrappedState}}$ must be of the form,

$$s_0\ \texttt{|}\ \texttt{while}\ t\ p \xrightarrow{+}_E s_1\ ,\ \texttt{k(while}\ t\ p\ \texttt{-> stop)} \xrightarrow{+}_E \ldots \xrightarrow{+}_E s_n\ ,\ \texttt{k(while}\ t\ p\ \texttt{-> stop)} \xrightarrow{+}_E s'$$

with $n \geq 0$, with s_i of sort `WrappedState`, $0 \leq i \leq n$, with $s_n\ \texttt{|}\ \texttt{t} \xrightarrow{+}_E s'$ and $E \vdash (\forall \emptyset)\ \texttt{evalTst}(s_n, t) = false$. In general we cannot guarantee that $E \vdash (\forall \emptyset)\ \texttt{evalTst}(s', t) = false$ holds, but in

the case we look at later, it will hold, as detailed there. Also, for $0 \leq i < n$,

- s_i , $\texttt{k(}t\texttt{ -> }p\texttt{ -> stop)} \xrightarrow{*}_E s_{i+1}$
- $E \vdash (\forall \emptyset)\ \texttt{evalTst}(s_i, t) = true.$

Observations and Lemmas (III)

Proof: By induction on the number n of different occurrences of expressions of the form s_i , `k(while` t p `-> stop)`, with s_i of sort `WrappedState`, $0 \leq i \leq n$, that appear in the sequence s_0 `|` `while` t $p \longrightarrow_E^* s'$. q.e.d.

Notation: given a program p, we use the notation p^n, working also for programs composed of an `Exp` t and `BlockStatements` p', $p = t\,|\,p'$, to mean:

- $p^0 = ;$
- $p^{n+1} = p\,|\,p^n$

We use composition with _|_ instead of composition with __ because the internal types in p need not match, i.e. $(t\,|\,p')^2 = (t\,|\,p'\,|\,t\,|\,p')$ and not $(t\,|\,p'\ t\,|\,p')$ where $p'\ t$ would be illegal.

Corollary: For any ground terms s_0 of sort `WrappedState`, t of sort `Exp`, and p of sort `BlockStatements`, if we have,

$$E \vdash (\forall \emptyset)\ s_0\ \texttt{|}\ \texttt{while}\ t\ p\ :\ \texttt{WrappedState}$$

then there is an $n \geq 0$ such that:

Observations and Lemmas (IV)

- $E \vdash (\forall \emptyset)\ s_0 \mid \texttt{while}\ t\ p = s_0 \mid (t \mid p)^n \mid t$
- $E \vdash (\forall \emptyset)\ \texttt{evalTst}(s_0 \mid (t \mid p)^i, t) = true,\ 0 \leq i < n$
- $E \vdash (\forall \emptyset)\ \texttt{evalTst}(s_0 \mid (t \mid p)^n, t) = false.$

Proof of Soundness of the Loop Rule

Theorem: The loop rule is sound.

Proof: We have to prove that the assumption

$$\{A \wedge (\texttt{evalTst(S,t)} = \texttt{true})\}\ \texttt{t | p}\ \{A\} \quad \{A \wedge (\texttt{evalTst(S,t)} = \texttt{false})\}\ \texttt{t}\ \{A \wedge (\texttt{evalTst(S,t)} = \texttt{false})\}$$

implies

$$(\dagger)\ \{A\}\ \texttt{while t p}\ \{A \wedge (\texttt{evalTst(S,t)} = \texttt{false})\}$$

By the semantics of Hoare triples applied to (†), we only need to show,

$$T_{\texttt{JAVAX}} \models (\forall V)\ A \wedge \texttt{(S|loop):WrappedState} \Rightarrow A\texttt{(S/S | loop)} \wedge \texttt{(evalTst(S | loop,t)} = \texttt{false}),$$

Proof of Soundness of the Loop Rule (II)

where, by definition, `loop` = `while t p`. Putting aside the termination assumption (`(S|loop):WrappedState`) we have,

(‡) $T_{\texttt{JAVAX}} \models (\forall V)\ A \Rightarrow A(\texttt{S/S | loop}) \wedge (\texttt{evalTst(S | loop,t)} = \texttt{false})$,

But, by definition of satisfaction of a universally quantified formula in an initial algebra, this is equivalent to proving, from the above assumption,

$(\forall s \in T_{\texttt{JAVAX,WrappedState}})\ T_{\texttt{JAVAX}} \models (\forall V - \{\texttt{S}\})\ A(\texttt{S}/s) \Rightarrow A(\texttt{S}/s\texttt{|loop}) \wedge (\texttt{evalTst}(s\texttt{|loop}, t) = \texttt{false})$

That is, for each $s \in T_{\texttt{JAVAX,WrappedState}}$, we have to show,

(♭) $T_{\texttt{JAVAX}} \models (\forall V - \{\texttt{S}\})\ A(\texttt{S}/s) \Rightarrow A(\texttt{S}/s\texttt{|loop}) \wedge (\texttt{evalTst}(s\texttt{|loop}, t) = \texttt{false})$.

Proof of Soundness of the Loop Rule (III)

By the termination assumption and the last corollary we know that, for each ground substitution $\theta : V - \{\mathtt{S}\} \longrightarrow T_{\Sigma_{\mathtt{JAVAX}}}$, if

$$(\star)\ \ T_{\mathtt{JAVAX}} \models (\forall \emptyset)\ (\theta(A(\mathtt{S}/s)),$$

then there is an $n \geq 0$ such that:

1. $E \vdash (\forall \emptyset)\ s\ \mathtt{|\ loop} = s\mathtt{|(t|p)}^n\mathtt{|t}$
2. $E \vdash (\forall \emptyset)\ \mathtt{evalTst}(s\mathtt{|(t|p)}^i, t) = true,\ 0 \leq i < n$
3. $E \vdash (\forall \emptyset)\ \mathtt{evalTst}(s\mathtt{|(t|p)}^n, t) = false.$

We also know that $(4) E \vdash (\forall \emptyset)\ \mathtt{evalTst}(s\mathtt{|(t|p)}^n\mathtt{|t}, t) = false$ holds if A holds. That is because (3) tells us that the test is false in that state and then with the second assumption we see that the test will stay false after being executed. At the positions where we

use (4) A holds.

Since (1) and (4) take care of the implication of the second conjunct in ($\flat$) (A is on the left-hand side of the implication), we have reduced the whole matter to showing that, if ($\star$) holds, then we must have,

Proof of Soundness of the Loop Rule (IV)

$$(\natural)\ \ T_{\texttt{JAVAX}} \models \theta(A(\texttt{S}/s\,|\,(\texttt{t}\,|\,\texttt{p})^n\,|\,\texttt{t})).$$

But this now follows by substitutivity from the first assumption and (1)–(3), by the chain of implications, where the last step requires (4) and the second assumption (A obviously holds).

$$T_{\texttt{JAVAX}} \models \theta(A(\texttt{S}/s)) \Rightarrow \theta(A(\texttt{S}/s)) \wedge \texttt{evalTst}(s,t) = true \Rightarrow \theta(A(\texttt{S}/s\,|\,\texttt{t}\,|\,\texttt{p})) \wedge \texttt{evalTst}(s\,|\,\texttt{t}\,|\,\texttt{p},t) = true \ldots$$

$$\Rightarrow \theta(A(\texttt{S}/s\,|\,(\texttt{t}\,|\,\texttt{p})^{n-1})) \wedge \texttt{evalTst}(s\,|\,(\texttt{t}\,|\,\texttt{p})^{n-1},t) = true$$

$$\Rightarrow \theta(A(\texttt{S}/s\,|\,(\texttt{t}\,|\,\texttt{p})^{n})) \wedge \texttt{evalTst}(s\,|\,(\texttt{t}\,|\,\texttt{p})^{n},t) = false \Rightarrow \theta(A(\texttt{S}/s\,|\,(\texttt{t}\,|\,\texttt{p})^{n}\,|\,\texttt{t}))$$

q.e.d.

Proving Hoare Triples in the Java+ITP Tool

The *latest* version (with support for this Java subset) of the ITP downloadable from

`http://banyan.cs.uiuc.edu/itp`

has an extension of the list of commands of the ITP specifically designed to support Hoare logic reasoning in our Java fragment.

To prove a Hoare triple

$$\{P\}\ \texttt{foo}\ \{Q\}$$

using this tool, one gives a `javax` command.

Proving Hoare Triples in the Java+ITP Tool (II)

Essentially, what the `javax` command does is to *translate* the Hoare triple goal into its semantically equivalent inductive theorem proving goal.

For example, a goal consisting of the Hoare triple

$$\{P\}\ \texttt{foo}\ \{Q\}$$

is translated into the (universally quantified) ITP goal

$$P\ \Rightarrow\ Q(S/(S\texttt{|}\ \texttt{foo}))$$

where S is the distinguished variable of sort `WrappedState`.

An Example

We can illustrate the use of the `javax` command with an absolute value program `absx`. We define the module `absx.maude` as follows:

```
fmod ABSX-JAVA is
including JAVAX .
op absolute : Int -> Int .
var I : Int .
ceq absolute(I) = I if I >= 0 = true .
ceq absolute(I) = - I if I >= 0 = false .
ops absx-init absx : -> BlockStatements .
eq absx-init = (int 'X ; int 'Z ;) .
eq absx = if (#i(0) <= 'X) ('Z = 'X ;) else ('Z = - 'X ;) .
endfm
```

Note that, in addition to defining `absx` and `absx-init`, we have also defined the `absolute` auxiliary function.

An Example (II)

The point is that using the `javax` command we can now give our desired Hoare triple directly to the ITP. We do so by: (1) loading `java-es.maude` and `absx.maude`; (2) loading `itp-tool.maude`; and (3) giving the `javax` command:

```
(javax ABSX-JAVA :
--- specification constants
(I:Int)
--- precondition
((S:WrappedState['X]) = (int(I:Int)))
--- program
absx-init
absx
--- postcondition
((S:WrappedState['Z]) = (int(absolute(I:Int))))
.)
```

An Example (III)

The ITP then responds by translating this Hoare triple into the semantically equivalent ITP goal:

```
==================================
label-sel: absx@0
==================================
A{I:Int}(
    (state(e(['X,any-loc(0)]['Z,any-loc(1)]),n(0),m([any-loc(0),int(
    any-int(0))][any-loc(1),int(any-int(1))]),out(noOutput))['X]
    = int(I:Int))
    ==>
    (state(e(['X,any-loc(0)]['Z,any-loc(1)]),n(0),m([any-loc(0),
    int(any-int(0))][any-loc(1),int(any-int(1))]),out(noOutput))
    |(if #i(0)<= 'X 'Z = 'X ; else 'Z = - 'X ;)['Z]
    = int(absolute(I:Int))))
++++++++++++++++++++++++++++++++++
```

An Example (IV)

Giving the auto command we then obtain:

```
Maude> (auto .)

==================================
label-sel: absx@0
==================================
(k(bool(0 <= I*Int)-> ?('Z = 'X ;,'Z = - 'X ;)-> stop),state(e(['X,any-loc(
    0)]['Z,any-loc(1)]),m([any-loc(0),int(I*Int)][any-loc(1),int(any-int(
    1))]),n(0),out(noOutput)))['Z]= int(absolute(I*Int))
++++++++++++++++++++++++++++++++++
```

An Example (V)

We can now give the following `split` command:

```
Maude> (split on (0 <= (I*Int)) .)

==================================
label-sel: absx@1.0
==================================
(k(bool(0 <= I*Int)-> ?('Z = 'X ;,'Z = - 'X ;)-> stop),state(e(['X,any-loc(
    0)]['Z,any-loc(1)]),m([any-loc(0),int(I*Int)][any-loc(1),int(any-int(
    1))]),n(0),out(noOutput)))['Z]= int(absolute(I*Int))
```

An Example (VI)

Using the `auto` command discharges this goal, leaving only the other identical goal in the split:

```
Maude> (auto .)

==================================
label: absx@2.0
==================================
(k(bool(0 <= I*Int)-> ?('Z = 'X ;,'Z = - 'X ;)-> stop),state(e(['X,any-loc(
    0)]['Z,any-loc(1)]),m([any-loc(0),int(I*Int)][any-loc(1),int(any-int(
    1))]),n(0),out(noOutput)))['Z]= int(absolute(I*Int))
```

An Example (VII)

Which is again discharged by the `auto` command:

```
Maude> (auto .)

q.e.d

++++++++++++++++++++++++++++++++++

Maude>
```